Planted-model evaluation of algorithms for identifying differences between spreadsheets

Anna Harutyunyan, Glencora Borradaile, Christopher Chambers, Christopher Scaffidi
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR, USA
{ harutyua, borradag, chamberc, scaffidc}@onid.orst.edu

***Abstract*—Users often need to test, debug or reuse spreadsheets. We present a new algorithm that can identify differences between two spreadsheets, providing a basis for future tools to help users compare two versions of a spreadsheet (thereby seeing what is new and needs testing) or two different spreadsheets (thereby seeing which is more appropriate for reuse in a situation). This algorithm, RowColAlign, is a two-dimensional generalization of the classic dynamic programming algorithm for solving the one-dimensional longest common subsequence problem. In addition, we present a new planted model for generating test cases to evaluate this algorithm and others like it, including the greedy SheetDiff algorithm presented in prior work. In our evaluation, our new RowColAlign algorithm made no errors at all on test cases, including test cases comparable to relatively large spreadsheets. Moreover, further analysis revealed that it is unexpected for our new algorithm to make errors except when spreadsheets contain an unrealistically small number of distinct values. These results are extremely encouraging, revealing our algorithm's potential as the basis for future spreadsheet tools.**

Keywords-human-centric computing; spreadsheets; analysis

I. INTRODUCTION

Spreadsheets are an important tool for collecting, storing, organizing, analyzing, and visualizing data [8][15]. Given the widespread importance of spreadsheets, researchers have adapted various software engineering models and methods to the spreadsheet domain, with the principal goal of helping people to implement spreadsheets more reliably (e.g., [11]). One problem yet to receive much investigation is how to help people *reuse* spreadsheets. This is a significant gap in our understanding because empirical studies have shown people sometimes choose among which spreadsheets to reuse [8][15].

This choice can be difficult because existing spreadsheet editors provide little support for comparing spreadsheets descended from a common ancestor. Organizations often generate numerous spreadsheets from templates that they retain and repeatedly instantiate [6][15]. Such versions need not be descended from one another in a linear ancestral relationship (X derived from Y derived from Z) but rather are often "siblings" or are even more distantly related. In situations like this, "track changes" and similar existing features are inapplicable. Instead, we require a tool whose input is two spreadsheets A and B and whose output is some information that would help a user understand the differences between A and B, to facilitate choosing whether to reuse A or to reuse B.

This work was sponsored by Google and NSF (CCF-0963921).

We have therefore begun to develop "spreadsheet-diff" algorithms that efficiently summarize the differences between spreadsheets. Prior work presented a first step in this direction by developing a candidate algorithm called SheetDiff, which was more accurate than existing commercial tools for summarizing differences between spreadsheets [7]. In that evaluation, the key concern of interest was how well the algorithm could recover the edits actually used to transform one spreadsheet into another. One limitation is that the algorithm relies upon greedy heuristics that sometimes lead to errors; specifically, we define an errors in this context as as an incorrect identification of the edits that actually took place (i.e., a failure to accurately recover the edits that were in fact used to transform one spreadsheet A to another spreadsheet B). Another limitation is that the algorithm sometimes goes into an infinite loop. In addition to this limitation in the algorithm, a limitation of the previous evaluation was that it relied on a small set of ten test cases; although these were based on real spreadsheets downloaded from the EUSES Spreadsheet Corpus [12], they did not provide an extensive empirical evaluation of how the algorithm would perform in a range of situations.

We now address these limitations by presenting a new dynamic programming algorithm called RowColAlign for identifying differences between spreadsheets, and we evaluate the accuracy of both RowColAlign and SheetDiff in a broader range of situations—including large spreadsheets, for which a spreadsheet-diff tool would be most beneficial. We create test case spreadsheet pairs using a novel planted model (a type of generative model) that covers the situations explored in the prior work, as well as many other situations.

In all test cases, we found our new algorithm made no errors. In contrast, our existing greedy algorithm made many errors. To account for these empirical results, we analyzed the algorithm's accuracy on large spreadsheets and found that it is not expected to make any errors except when spreadsheets contain far fewer distinct values than is reasonable to expect in most real spreadsheets. Therefore, RowColAlign offers advantages over existing algorithms and provides the basis for a new generation of tools that can accurately recover edits to aid spreadsheet testing, debugging and reuse.

In addition, our work demonstrates a testing method that has not yet been applied to spreadsheet-related problems. This method could be used to evaluate whether other algorithms outperform our own. We anticipate that our experiences may provide researchers with methodological ideas about how to test algorithms of their own for other problems.

II. Related Work

Empirical studies highlight several reasons why people might value tools that can describe differences between spreadsheets. One reason is that studies show users frequently create spreadsheet errors [16], and although various tools and techniques support testing and debugging (e.g., [1][5][9] [11][17]), actually testing and debugging takes time. If users had the ability to identify what portions of a spreadsheet had changed since the last round of testing, they could focus on testing and debugging in those areas. Moreover, for the small but increasing number of organizations that audit spreadsheets [6], auditors could likewise focus on areas of spreadsheets that changed since the last audit—which would be valuable, since auditors typically miss over 20% of errors in spreadsheets [16], often due to the inability to focus attention on the right areas of spreadsheets [6]. Finally, users often want to reuse spreadsheets but need to choose among multiple options [8]. They often choose a buggy spreadsheet to reuse and, as a direct result, unintentionally propagate errors [6]. Thus, comparing differences between spreadsheets could also facilitate choosing which to reuse.

In response to this need, several tools have become available. The most basic of these are TellTable [2] and Microsoft Excel's "Track Changes" feature, which highlight cells, rows and columns edited. Such tools can show changes made since the last save event or since the last time changes were accepted, but they cannot display changes between two arbitrary spreadsheets during reuse; for example, if two users each edited a spreadsheet O to create A and B, respectively, then a third user could not use these features to compare A and B when choosing one to reuse. More appropriate tools for this task include DiffEngineX [13], Synkronizer [19], and Suntrap Excel Diff [18], which can compare arbitrary spreadsheets. For each tool, users can specify a pair of spreadsheets. The tool then computes the difference between these spreadsheets (as a set of modified/inserted/deleted rows, columns, and cells) and displays changes as colored regions. For example, as shown in **Error! Reference source not found.**, Synkronizer highlights inserted rows and columns (column B and row 3 of spreadsheet B) and edited cells (cell B2 in spreadsheet A and cell C2 in spreadsheet B.) (Deleted rows or columns, as well as cells with format changes would be highlighted in various colors.)

Unfortunately, Synkronizer's output even in this simple case is inaccurate. The actual edits used to turn spreadsheet A into spreadsheet B were to insert B's second column ("c", "g", …), to insert B's second row ("d", "d", "d"), and to change B's A3 from "d" to "e". Synkronizer instead inferred that B's row 3 is new, and that C2 was edited. Prior empirical evaluation showed that these tools often made such errors, incorrectly identifying up to 10-20% of row and column changes [7]. The SheetDiff algorithm (described below in detail) performed a bit better, with an error rate of 6% on row and column edits.

Conceptually it might seem as if such inaccuracy does not matter, as if all ways of accounting for spreadsheet differences are equally valid, but considering the potential applications of such an algorithm reveals that such inaccuracy can matter in practice. For example, an auditor might find an error in a certain cell, and management might need to determine which user created that error; identifying the wrong cell could lay the blame for the error at the wrong user's feet. As another example, highlighting the wrong rows when helping a user with testing would focus attention on the wrong area to test.

Aside from the "spreadsheet-diff" tools above, there has been a small amount of relevant theoretical work on algorithm performance (rather than accuracy). The problem of summarizing spreadsheet changes includes a requirement to find a subsequence of rows and columns that have been changed—or, equivalently, finding a subsequence of rows and columns that have not been changed. Identifying a *maximal* subsequence of unchanged rows and columns (so as to minimize the differences eventually shown to the user) essentially constitutes a 2-dimensional generalization of the classic longest-common-subsequence (LCS) problem. LCS in one dimension has a classic, efficient dynamic program [4], but multi-dimension LCS is NP-hard [3]. So in the worst case, no algorithm can find the maximal common subsequence of spreadsheet rows and columns in less than exponential time (unless P=NP).

Nonetheless, spreadsheet edits need not always be worst-case in practice. All of the tools above, as well as our own algorithm, typically run in a few seconds on spreadsheets from the EUSES Spreadsheet Corpus. Consequently, in our current paper, we primarily focus on the *accuracy* of algorithms, relative to *real* edits on a subsequence of rows, columns and cells, rather than algorithmic complexity.

III. Algorithms for Identifying Spreadsheet Changes

Motivated by the spreadsheet application mentioned above, we present two algorithms aimed at accurately identifying the edits used to transform one spreadsheet into another. Specifically, each algorithm's input is a pair of spreadsheets (A, B), and its output is a list of edits consisting of row or column insertions or deletions, or edits to individual cells.

A. Algorithm from prior work: SheetDiff

SheetDiff is a greedy, iterative algorithm based on the idea of finding edit operations that each transform A into B as much as possible. We have previously defined the algorithm in detail [7] and summarize it here. SheetDiff, takes A and B and selects the one edit on A that makes it more similar to B according to a similarity metric. It applies this edit and then repeats the comparison to B until no further improvements can be found.

Specifically, SheetDiff compares A and B to find the row r and the column c that contain the highest number of different cells. It computes four spreadsheets (A − r, A − c, A + r, A+c) formed by deleting r or c from A, or by inserting B's version of r or c into A (or, equivalently, deleting r and c from B). Of these four options, it chooses the one that minimizes the number of cells that remain different relative to B. This choice effectively selects a row or column insertion or deletion (which SheetDiff appends to its output, a list of edit operations) and also yields a spreadsheet A' that is more similar to B than A was. SheetDiff then uses this spreadsheet A' for the next iteration, and the algorithm continues iterating until achieving a similarity metric threshold between A and B. It then computes all remaining non-identical cells by pairwise comparison and appends an edit operation for each differing cell.

B. New algorithm: RowColAlign

1) Idea behind the algorithm

RowColAlign is a new dynamic-programming algorithm based on the idea of finding maximal subsequences of rows and columns that are *nearly* unchanged in A relative to B. RowColAlign takes A and B and first attempts to find what they have in common; we refer to this *common subsequence of rows and columns* as the "target alignment" of A and B. Everything that they do not have in common must be an edit and therefore must appear in the output list (i.e., the edits that transform A into B), either as an insertion if the row/column in question is in B but not A, or a deletion if the row/column is in A but not B. A minor detail is that the areas in common need not be identical; a small number of cell-level differences are tolerated and represented as cell-level edits.

The target alignment T(A,B) of A relative to B is a subsequence of rows and columns. The target alignment of one spreadsheet to another is also a spreadsheet, consisting of some rows and columns from the starting spreadsheets. For example, Figure 2 depicts the target alignment T(A,B) of A relative to B when one row and column have been deleted from A (shaded), two rows and columns have been inserted (shaded in B), and three additional cells have been edited (shown as gray numbers in T). T(B,A) is the target alignment of B relative to A. T(A,B) and T(B,A) always have the same number of rows and columns, but because they are subsets of rows and columns from A and B, respectively, T(A,B) and T(B,A) differ in cells that contain remaining cell-level changes. In other words, the shape of T(A,B) and T(B,A) are always identical, but they may differ in their contents. When we are merely interested in the rows and columns of A and B that appear in the target alignment but are not interested in the specific cell-level values that appear, we will refer to T(A,B) and T(B,A) as just T.

Having framed the problem as one of finding a target alignment T, we note that this can be thought of purely in terms of row and column deletions: that is, identifying the rows and columns that need to be deleted from A and B to leave behind T(A,B) and T(B,A), respectively. Treating the problem as one of deletions essentially abstracts away from the question of whether A needs to be turned into B, or whether B needs to be turned into A; once a target alignment is found, a tool could use this T to show the user what edits would turn A into B *or* vice versa. Consequently, in the following discussion, we present our algorithm in terms of deleting rows and columns to generate a target alignment.

2) Details of the algorithm

Our algorithm has four phases. In the first, it compares A and B to identify which rows should be deleted. In the second phase, it compares A and B to identify which columns should be deleted. In the third phase, it deletes the rows and columns, then performs pairwise comparisons on the remaining individual cells inside T(A,B) and T(B,A). Finally, if any row or column pair in T remains mostly different, it is represented as a row or column edit, rather than a list of cell-level edits.

In the first phase, given a pair of spreadsheets A and B consisting of rows $\langle A_1, A_2, \ldots, A_m\rangle$ and $\langle B_1, B_2, \ldots, B_n\rangle$, the algorithm seeks a target alignment consisting of two respective

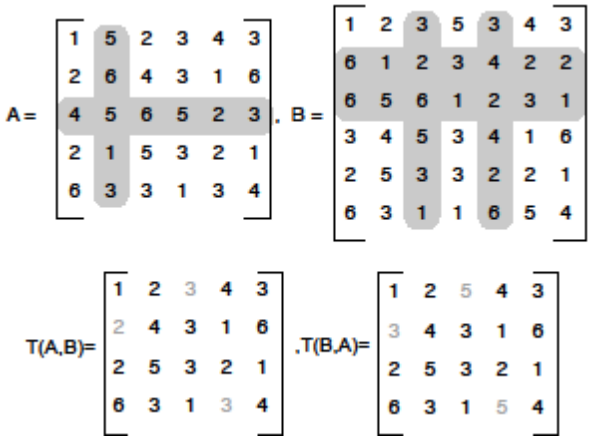

Figure 1. Target alignments of spreadsheets; shading indicates where rows and columns differ by insertion/deletion, and grey numbers indicate where individual cells differ due to edits.

subsequences of rows, $\langle T(A,B)_1, T(A,B)_2, \ldots, T(A,B)_t\rangle$ and $\langle T(B,A)_1, T(B,A)_2, \ldots, T(B,A)_t\rangle$. These subsequences should be chosen so T(A,B) and T(B,A) contain few cell-level edits; that is, for each row k in T(A,B), the corresponding row in T(B,A) should be very similar. In particular, our algorithm selects the subsequences that maximize the expression

$$\text{Score}(T) = \sum_{k=1}^{t} \text{LCS1D}(T(A,B)_k, T(B,A)_k) \quad (1)$$

where we define LCS1D(x,y) to be the length of the longest common subsequence of cells in two rows x and y. The value of LCS1D(x,y) can be computed with a standard one-dimensional LCS algorithm using dynamic programming (i.e., where the rows are strings, and the cells are characters in the strings). If two rows are identical, then LCS1D would return their length; if they have no cells in common, then LCS1D would return 0. Thus, making good selections for T(A,B) and T(B,A) will lead to choosing many rows for which LCS1D returns large numbers, resulting in a high total Score.

In a sense, formalizing the problem in this fashion treats each spreadsheet as a sequence of rows. Rows, and subsequences of rows, should be included if they contribute the most to the total Score(T). This is similar to the problem structure in one-dimensional LCS, except that here the rows are playing the role of "characters" and the spreadsheet is a "string" containing a sequence of rows/characters. To illustrate, consider a situation where A and B each have only 1 column, so rows x and y each have 1 cell. Then LCS1D(x,y) is 1 if the cell in x and the cell in y contain the same value, and 0 otherwise. Thus, in this illustration, finding the optimal target alignment requires finding a maximal subsequence of identical cells, which is precisely the one-dimensional LCS problem. In our more general case, we allow for multiple columns, so some rows might only partially match, allowing for other values of LCS1D. Although efficiently making a maximal choice in two dimensions is likely not possible (because it is NP-hard, as discussed earlier), we can decouple the dimensions and focus on finding optimal rows, then finding optimal columns, by generalizing the one-dimensional LCS algorithm. The result might not be a *jointly maximal* number of rows and columns overall, but our hypothesis is this approach will still accurately recover the edits performed in practice.

Thus, our algorithm proceeds in the same fashion as the classic LCS dynamic programming algorithm. Given a row sequence $X=<X_1, X_2, \ldots, X_m>$, we define its row-wise prefix of length i to be $X^i = <X_1, X_2, \ldots, X_i>$. We define S(i,j) to be the maximal value of Score that can be obtained considering only prefixes A^i and B^j. When considering whether some row i of A and row j of B should be included in T (as in the one-dimensional LCS algorithm), there are three possibilities:

- A_i and B_j might both be included in T, in which case S(i,j) includes a contribution from LCS1D(A_i,B_j)
- A_i might not be included in T(A,B). In that case, it is marked as deleted and S(i,j) must equal the maximal Score that can be still obtained using A^{i-1} and B^j, which is S(i-1,j).
- B_j might not be included in T(B,A). Analogously to the second case, in this case S(i,j) == S(i,j-1).

The optimal selection is the one that maximizes S(i,j) by choosing among these three options. (There is no need to explicitly consider the fourth possibility of neither row being included in T, as this is equivalent to first choosing the second option and then, in the next step, choosing the third option.)

This leads to the algorithm, AlignR, shown in Figure 3. We also define an analogous algorithm, AlignC, that is identical to AlignR except that it operates on columns rather than rows. (In practice, we have implemented a single algorithm, Align, that accepts the two spreadsheets as well as third parameter indicating whether to perform row-wise or column-wise alignment.) The final RowColAlign algorithm runs AlignR to compute what rows to delete, runs AlignC to determine which columns to delete, deletes the rows and columns, then compares cells pairwise to identify cell-level edits. This output is post-processed for conciseness: cell-level edits are simply listed in the diff or, if at least n(1-2p-q) of the cells in a row or column are different, then the entire row or column is listed as an insertion/deletion in A and B (i.e., it is deleted from T).

Algorithm AlignR

Inputs:

- Spreadsheet A, a sequence of m rows
- Spreadsheet B, a sequence of n rows

Output: T(A,B) and T(B,A)

Algorithm:

```
for i = 0 to m
    S(i,0) = 0
for j = 0 to n
    S(0,j) = 0
for i = 1 to m
    for j = 1 to n
        S(i,j) = max(S(i-1,j); S(i-1,j-1) + LCS1D(Ai, Bj); S(i,j-1))
Let i = m and j = n
while i ≥ 1 and j ≥ 1
    if S(i,j) == S(i-1,j-1) + LCS1D(Ai, Bj)
        include Ai and Bj in T(A,B) and T(B,A), respectively
        i = i – 1
        j = j – 1
    else if S(i,j) == S(i-1,j)
        make a note that Ai is not in T(A,B), i.e., mark as deleted
        i = i -1
    else
        make a note that Bj is not in T(B,A), i.e., mark as deleted
        j = j – 1
return T(A,B) and T(B,A)
```

Figure 2. Algorithm to select rows for inclusion in T

IV. Empirical evaluation with corpus spreadsheets

Prior work manually mined the EUSES Spreadsheet Corpus [12] for 8 pairs of spreadsheets that appeared to be respective versions of one another [7]. Inspecting these spreadsheets revealed the percentages of cells, rows and columns that appeared to have been changed; of course, it was not possible say for certain what the *true* set of edits performed actually had been, for testing whether spreadsheet-diff algorithms accurately recovered edits. Therefore, 10 spreadsheets were randomly selected from the corpus and manually edited with the same kinds and quantities of edits that appeared to have been performed on the first 8 spreadsheet pairs. This yielded 10 test cases resembling real-world spreadsheet pairs, and for which the true series of edits was known.

Running SheetDiff on these 10 test cases in the prior study revealed that it made errors on 60% of the test cases (spreadsheets). The actual errors were fairly small: on a per-cell basis, the algorithm misclassified only 2% of the cells as incorrectly modified or not modified, and on a per-row or per-column basis, the algorithm misclassified only 6% incorrectly. Thus, the algorithm made relatively local errors but made these on over half of the tests.

In the current study, we used the same test cases to evaluate our new RowColAlign algorithm. We found that it made no errors at all. In every case, it correctly identified the edits to rows, columns and cells we had modified. Thus, at least on the 10 test cases previously created by hand, we found that the new algorithm was consistently more accurate than the old.

V. Empirical evaluation with a generative model

Extending our empirical evaluation to a broader range of situations, we measured the accuracy of both algorithms using spreadsheet pairs generated with a novel planted model.

A. Planted model

Systematically evaluating our algorithms required a different approach than manually searching through the EUSES Corpus for spreadsheets that look similar, or manually creating test cases from spreadsheets we find. Manual test generation as in our prior work is time-intensive and limits the number of situations that can be explored. Moreover, manual test generation introduces a potential for unintentional bias.

Therefore, we have developed a planted model for generating test cases. Such a model is called “planted” because it provides a “correct” solution, from which it generates a problem that the algorithm is supposed to solve. Thus, the problem (test case) is “planted” in a known solution.

Our model for generating test cases simulates the situation where an original spreadsheet O is modified by two different people to create spreadsheets A and B. The algorithm's responsibility is to recover the differences between A and B.

We generate test cases as follows. We fix the original spreadsheet O as n x n cells. O is populated by letters drawn uniformly at random from the alphabet {1, 2, …, s}. We create A and B by copying O. Then, we delete each row and column of A with probability p, then delete each row and column in B with probability p. (Since the ultimate purpose is to test algorithms that compare A and B, there is no need to randomly insert rows and columns because inserting a row into B would be equivalent to deleting that row in A, and vice versa.) Finally, we then randomly overwrite each cell in B with probability q using a new value drawn randomly from our alphabet.

This model for generating test cases also provides a known solution to each problem comparing A and B. The correct target alignment T consists of those rows and columns that were not deleted when generating A and B, and the correct list of cells that were modified is also tracked.

B. Evaluation

We evaluated our algorithms by using our planted model to generate test cases, with a range of values for parameters: spreadsheet width and height n, alphabet size s, probability of row and column deletion p, and probability of cell-level edit q. To create moderate-difficulty test cases, we selected parameter values that covered the approximate range we observed in our EUSES corpus test cases (Section IV). We then challenged our algorithms with even more difficult test cases, targeting the upper quartile of typical EUSES spreadsheets in terms of size.

1) Range of moderately difficult test cases

In the EUSES corpus test cases, we observed that the spreadsheet area varied in the range of 90-3212 cells (i.e., equivalent n from 9.5-56.7), alphabet size in the range 50-671, p in the range 0.0167-0.08, and q in the range 0.0016-0.05 (based on the percentages of rows and columns that were inserted or deleted, and the number of remaining cells that were edited, in the test cases). We used these as approximate ranges to generate test cases. For each parameter setting, we generated 25 test cases and then computed error rate as the number of test cases for which the algorithm made any error at all in identifying which rows, columns, or cells had been edited.

We found SheetDiff algorithm made errors on some spreadsheets (Figure 4), with four suggestive patterns. First, the error rate was flat at 0% as spreadsheet width n rose into the higher end of its range. Second, increasing the rate of row and column changes p noticeably increased the error rate. Third, the error rate was generally flat over increasing cell-level edit rate q but eventually rose at q=0.401. Fourth, the error rate was flat at 0% as s rose into the high end of its range. Overall, these error rates are widely-scattered, indicating that unpredictability is the most consistent trait of SheetDiff, rather than any dominant trends over the parameter space.

These results contrast with those for our new RowColAlign algorithm. For this dynamic programming algorithm, we did not observe any errors at all for any of the parameter settings above. In every case, the algorithm always recovered precisely the correct target alignment and cell-level edits.

We informally examined what RowColAlign was doing internally that allowed it to find the edits so accurately. We found that in many cases, when the algorithm invoked LCS1D

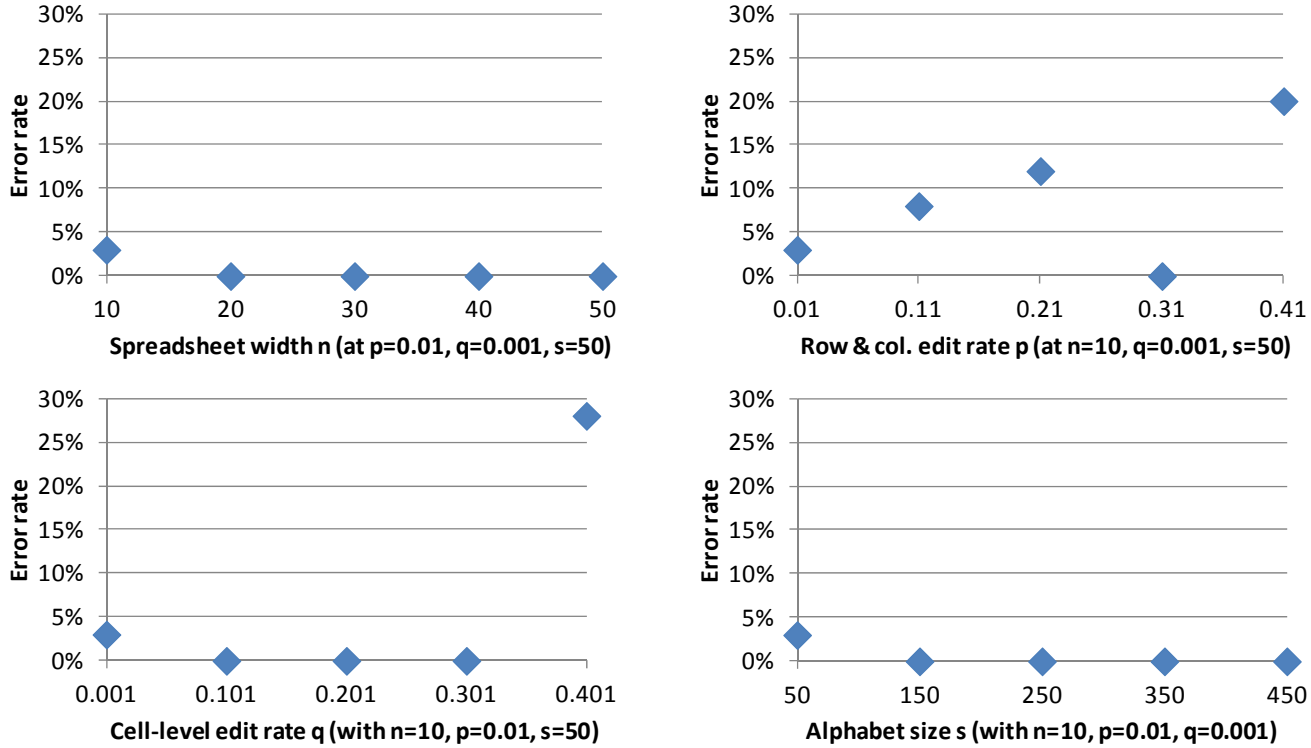

Figure 3. Error rates for SheetDiff on test cases generated by our planted model. Our new RowColAlign made no errors.

to compare a row of A with a row of B, it was rare that the rows generated a high LCS1D unless if, in fact, the rows actually originated from the same row in O. In other words, it was rare that two rows happened to have similar values solely by chance, so using LCS1D to detect row-wise similarity rarely led to misalignments between rows. (The same was true column-wise.) In Section VI, this informal observation provides the key insight required for theoretical analysis of RowColAlign's accuracy.

2) Extremely difficult test cases

To further challenge RowColAlign, we tried it with atypically large spreadsheets, with n=100 for an effective area of 10000 cells. By way of comparison, the top quartile of spreadsheets in the EUSES corpus begins at 961 input cells (effective n=31) [12], so our test cases were well into the extremely high end of this distribution. We set p and q at the maximums (0.08 and 0.05, respectively) demonstrated by the corpus-based test cases from Section IV.

Given our observation above that RowColAlign's accuracy appeared to depend on ensuring that no two rows happen to have similar values solely by chance, we tried to cause errors by reducing the alphabet size, so that many more cells would contain the same values. In the EUSES Corpus test cases, s very roughly rose linearly with the spreadsheet area, with approximately 1 distinct value for every 3 cells. To challenge the algorithm, we considered test cases that had 1 distinct value for every 10 cells ($s=n^2/10=1000$), 100 cells (s=100), and 1000 cells (s=10). For each value of s, we generated 25 test cases.

On these more difficult test cases, our existing SheetDiff demonstrated extremely high error rates (Table 1). Even with only 10 cells per distinct value (s=1000), the error rate was 52%. Once the number of cells per distinct value increased to 1000 (s=10), SheetDiff made errors on 96% of the test cases.

In contrast, as with our moderately difficult test cases, RowColAlign did not make any errors even on these more difficult test cases.

VI. Analytical Evaluation of RowColAlign

To investigate the cause for these empirical results, we analyzed the algorithm's accuracy as spreadsheets become exceptionally large. In addition, we investigated the asymptotic algorithmic complexity (runtime) as a function of spreadsheet size.

A. Accuracy of RowColAlign

We observed during our evaluation that it was rare for two rows to be similar solely by chance, leading to misalignment. In other words, rows tended to have only short subsequences of cells in common if they in fact were not supposed to be aligned. In contrast, rows that *should* have been aligned tended to have relatively long subsequences in common.

Table 1. Error rates for large spreadsheets (n=100) with many changes (p=0.08, q=0.05) and relatively small alphabet sizes.

Alphabet size s		Error rate	
s	n^2/s (approx. # cells per distinct value)	SheetDiff	RowColAlign
1000	10	52%	0%
100	100	60%	0%
10	1000	96%	0%

We formalize these notions below. First, we show that if a row from A is in T(A,B), then it is expected to be aligned by RowColAlign with the correct row from B in T(B,A), unless if the alphabet size is unrealistically small. Second, we show that if a row from A should appear in T(A,B), then it is indeed expected to be placed by RowColAlign into T(A,B).

Putting these results together, the rows that should appear in T(A,B) are expected to appear there in the right positions.

1) Rows in T(A,B) are expected to be properly aligned

Consider what would be required for a row x in T(A,B) *not* to be properly aligned. In that case, x is aligned with some row y' in T(B,A) other than the correct row y from B. This can only happen if LCS1D(x,y') exceeds LCS1D(x,y).

The incorrect alignment would involve two rows that are similar only by chance. For two strings of length u whose characters are independent and identically distributed from an alphabet of size s, the expected length of the longest common subsequence is known to be between $u/\sqrt{s}$ and $ue/\sqrt{s}$ (where e is the base of the natural logarithm) [10]. The rows (or columns) compared by LCS1D will generally not have n cells, due to deletions when A and B are generated from O. Specifically, due to these deletions, u has an expected value $(1-p)n$. Therefore, LCS1D(x,y') is expected to return at most $(1-p)ne/\sqrt{s}$.

In contrast, the correct alignment would involve two rows that are nearly identical but will differ somewhat due to column deletions and cell-level edits. For each cell, the probability is $(1-p)(1-p)(1-q)$ that it is not deleted in A, not deleted in B, and not edited in A when the test case was generated. Thus, LCS1D(x,y) is expected to return $(1-p)^2(1-q)n$ (or slightly higher, due to chance similarity between x and y).

Thus, a row in T(A,B) is expected to align with some row other than the correct row in T(B,A) only if

$$(1-p)ne/\sqrt{s} \geq (1-p)^2(1-q)n \quad (2)$$

Or

$$\sqrt{s} \leq {(1-p)e}/{(1-p)^2(1-q)} \quad (3)$$

Unless p or q rises toward 1 (meaning that A and B entirely differ), constraint (3) requires s to be extremely small. For p=0.08 and q=0.05, at the limit of what we observed in EUSES corpus test cases, there would need to be fewer than 10 distinct values to cause any x in T to be misaligned. Moreover, because the expected value of LCS1D for two random strings of length u (above) was *between* $u/\sqrt{s}$ and $ue/\sqrt{s}$, the estimate above is *worst-case*; dropping the factor of e would reduce the threshold below 2, meaning that all cells would need to be the same. Thus, it is generally expected for each row in T(A,B) to be properly aligned. This result is independent of spreadsheet size.

(As a side note, even though the argument above based on expected values indicates that incorrect alignment of two rows is not expected, it is impossible at present to precisely compute *the probability* that two rows will be incorrectly aligned. The

reason is that it remains an open theoretical problem to determine the probability distribution for the expected length of the longest common subsequence between random strings.)

2) It is unexpected for a row that should be in T to be omitted

Consider what would be required for a row x that *should* be in T(A,B) to be omitted. RowColAlign will not simply omit x; doing so would decrease the total Score in equation (1). Rather, the only reason RowColAlign might omit x from T is if that omission makes it possible instead to include some *other* row or rows from A that do not belong in T but together contribute enough to the Score to make up for losing x. Thus, RowColAlign has a choice between including one or more rows that should not be included in T(A,B) or including x.

Figure 5 illustrates this scenario, where row $x=A_3$ might be omitted in preference to other rows. A_3 should be matched to row B_2, as indicated by the solid line. However, RowColAlign could instead select A_1-B_3 and A_2-B_4. Including these *and* the correct choice is not possible: subsequence $<A_1, A_2, A_3>$ cannot be matched to $<B_3, B_4, B_2>$ because the latter would not be a valid subsequence of B, due to misordering of B_2 relative to the others. Visually, this misordering shows up as intersections: the dashed lines must *cross* the solid line if alternatives are to be considered in place of the correct choice.

Thus, for x to be omitted, the total summed similarity from r incorrect row alignments (due to chance) must exceed the similarity of one correct row alignment:

$$r(1-p)ne/\sqrt{s} \geq (1-p)^2(1-q)n \quad (4)$$

Or
$$s \leq \left[e/(1-p)(1-q)\right]^2 r^2 \quad (5)$$

As noted above, each of the r incorrect rows selected from A should not actually be in T(A,B). Therefore, they must be rows that were deleted from B (they only exist in A). Moreover, they must be matched up with rows in B that should not appear in T(B,A) (they only exist in B). There are expected to be, at most, pn such rows in A and B each. Of these, only half of the possible incorrect matches are expected to conflict with any correct match (i.e., having dashed lines that cross the solid line in an illustration like Figure 5). Thus,

$$s \leq \left[e/(1-p)(1-q)\right]^2 pn/2 \quad (6)$$

Or
$$n^2/s \geq \left[2(1-p)(1-q)/ep\right]^2 \quad (7)$$

Consequently, a row x that should be in T is not expected to be omitted unless the ratio of area to distinct alphabet values exceeds a threshold. For example, as in our difficult test cases with p=0.08 and q=0.05, we would not expect any errors until $n^2/s \geq 64$. Once again, because the expected value of LCS1D for two random strings of length u (above) was *between* $u/\sqrt{s}$ and $ue/\sqrt{s}$, the estimate above is *worst-case*; dropping the factor of e would raise the threshold to $n^2/s \geq 477$.

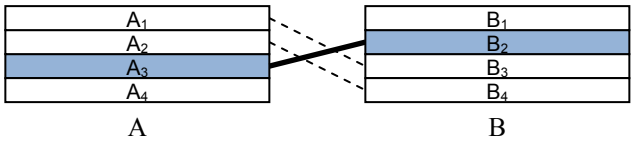

Figure 4. Notional depiction of what would be required for a correct row match (solid line between shaded boxes) to be omitted from the target alignment: some incorrect match(es) (dashed line(s)) must occur instead.

Empirically, we did not observe any errors even when the ratio reached 1000. A possible reason is that the threshold above can only be reached in an unusual circumstance: many rows that only exist in A must be highly similar matches to many rows that only exist in B, and these matches must all happen to be located in A and B so that they conflict with a correct choice. Because this is plausible but did not occur empirically, we anticipate infrequent errors to occur near or above the upper end of the range we explored, at area/alphabet ratios far beyond those in our EUSES test cases.

B. Complexity of RowColAlign

We chose not to explore larger values of n (and n^2/s) due to the runtime of RowColAlign, which in the worst case is quadratic in the area of the spreadsheets. This can be concluded in three steps. First, the standard dynamic programming algorithm for LCS1D is $O(u_A u_B)$, where u_A and u_B are the lengths of the two string inputs to LCS1D; in our case, these lengths are O(n), so LCS1D is $O(n^2)$. Second, LCS1D is invoked by AlignR within a loop over the rows of B inside a loop over the rows A, so AlignR is $O(n^4)$. Third, the complexity of AlignC is the same by analogy, and RowColAlign simply invokes AlignR and AlignC. Thus, RowColAlign is $O(n^4)$, or quadratic in the input area.

This is better than the worst case asymptotic algorithmic complexity for SheetDiff, which we have found occasionally fails to terminate. The reason is it sometimes makes a row or column edit and then, in a subsequent step, makes another edit that cancels out the first edit (e.g., row insertion, then deletion). This problem, in turn, is due to the fact that when SheetDiff selects a row or column to insert or delete, this does not necessarily improve similarity between spreadsheets; SheetDiff chooses the best available edit, even if that edit reduces similarity. We could trivially modify SheetDiff to terminate when no similarity-improving edits are available, but in some cases it would then fail to find needed edits at all, leading to a higher error rate. The fundamental issue is that similarity is not always a monotonically increasing function as successive edits transform one spreadsheet into another. Therefore, a greedy, local optimization algorithm such as SheetDiff cannot be guaranteed to terminate *and* be guaranteed to find the right answer every time. A dynamic programming (such as RowColAlign) is better suited to the problem at hand.

VII. DISCUSSION

We have presented a new algorithm for computing the differences between two spreadsheets. This algorithm, RowColAlign, is essentially a two-dimensional generalization of the dynamic programming algorithm for solving one-dimensional LCS. In addition, we have presented a new planted model for generating test cases to evaluate this algorithm and others like it, including the greedy SheetDiff algorithm presented in prior work. Our planted model was effective at

generating test cases that spanned the situations covered by prior work. Moreover, this planted model successfully uncovered differences between the accuracies of SheetDiff and RowColAlign.

In particular, unlike SheetDiff, our new RowColAlign algorithm made no errors at all on test cases comparable to typical spreadsheets, nor on test cases using large generated spreadsheets. An analysis revealed that, due to the unlikelihood of rows being similar by chance, it is unexpected for RowColAlign to make errors except when spreadsheets contain an unrealistically small number of distinct cell values. To facilitate replication of our work and testing on other algorithms, we did not retain our test cases, but we can share the code for generating test cases.

A. Limitations

As in our prior work, we have focused here on row insertion, row deletion, and cell-level edits. Spreadsheet editors support other operations, such as copy-paste (where affected regions might not subtend an entire row or column) as well as cell-fill (where a cell formula is dragged over and applied to other cells). We have not evaluated how our algorithms would perform when faced with spreadsheets pairs where such edits have occurred, nor does our planted model generate these edits. Further investigation will be required to explore the extent to which such modifications occur as well as how much they interfere with our algorithms' accuracy. Moreover, we did not consider spreadsheets with cells containing formulas, which should be addressed as well. Future work could also investigate how much tools based on our algorithms facilitate testing, debugging, and reuse of spreadsheets.

B. Implications for tool development

Our empirical results suggest that RowColAlign could serve as a reliable algorithm driving a new generation of spreadsheet-diff tools. Such tools might be used to highlight differences between spreadsheets to focus attention during testing and reuse. As another potential application, tools could mine repositories of spreadsheets to find pairs that are very similar to one another (i.e., for which the target alignment T encompasses nearly the entire spreadsheet). Then, if a formula error is found in one spreadsheet, owners of similar spreadsheets could be notified of the error so that they can repair their own. With additional new algorithms to complement RowColAlign, such tools could even cluster spreadsheets by similarity and abstract clusters into templates that could be provided to help people create new spreadsheets.

One significant potential challenge that we foresee is that RowColAlign's asymptotic algorithmic complexity is quadratic in spreadsheet area. The algorithm generally ran in a few seconds or minutes on each test case of our study, but the largest spreadsheet in the EUSES Spreadsheet Corpus apparently has over *3 million* cells [12]. While such enormous spreadsheets are extremely rare, perhaps they would be more common if users had effective tools for understanding them. For example, not only might a tool for comparing large spreadsheets to one another be useful, but a tool could also be useful for comparing large *regions* within a spreadsheet to one another (if those regions are supposed to be somewhat similar to one another, in terms of formulas used).

To improve the performance of RowColAlign so it could be used in tools for massive spreadsheets, one possibility might be to parallelize RowColAlign, as other researchers have done to improve the one-dimensional dynamic programming algorithm for LCS so that it can be applied to genetic sequences [14]. Parallelized operation would be a natural fit to the cloud-based computing platforms such as Google Docs where spreadsheet editors are now being deployed. Moreover, integrating tools based on our new algorithm into these platforms would provide a means for studying the extent to which these tools help users to compare and understand spreadsheets.

REFERENCES

[1] Abraham, R, and Erwig, M. (2005) Goal-directed debugging of spreadsheets. *VL/HCC*, 37-44.

[2] Adler, A, Nash, J, and Noel, S. (2004) TellTable: A server for collaborative office applications. *Computer Supported Cooperative Work (CSCW)*, 6-10.

[3] Amir, A, Hartman, T, Kapah, O, Shalom, B, and Tsur, D. (2007) Generalized lCS. *Proceedings of the 14th International Conference on String Processing and Information Retrieval*, 50-61.

[4] Bergroth, L, Hakonen, H, and Raita, T. (2000) A survey of longest common subsequence algorithms. *Seventh International Symposium on String Processing and Information Retrieval*, 39-48.

[5] Bishop, B, and McDaid, K. (2008) Unobtrusive data acquisition for spreadsheet research. *VL/HCC*, 139-142.

[6] Caulkins, J., Morrison, E., and Weidemann, T. (2007) Spreadsheet errors and decision making: Evidence from field interviews. *Journal of Organizational and End User Computing 19*, 3, 1–23

[7] Chambers, C, Erwig, M, and Luckey, M. (2010) SheetDiff: A tool for identifying changes in spreadsheets. *VL/HCC*, 85-92.

[8] Chambers, C, and Scaffidi, C. (2010) Struggling to excel: A field study of challenges faced by spreadsheet users. *VL/HCC*, 187-194.

[9] Chintakovid, T, Wiedenbeck, S, Burnett, M, and Grigoreanu, V. (2006) Pair collaboration in end-user debugging. *VL/HCC*, 3-10.

[10] Chvatal, V, and Sankoff, D. (1975) Longest common subsequences of two random sequences. *Journal of Applied Probability*, 306-315.

[11] Erwig, M. (2009) Software engineering for spreadsheets. *IEEE Software*. *29*, 5, 25-30.

[12] Fisher II, M, and Rothermel, G. (2005) The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. *WEUSE 2005*. 47-12.

[13] Florencesoft. (2012) *DiffEngineX — Compare Excel Worksheets*. http://www.florencesoft.com/

[14] Liu, W, Chen, L, and Zou, L. (2007) A parallel LCS algorithm for biosequences alignment. *Proceedings of the International Conference on Scalable Information Systems*, 1-8.

[15] Nardi, B. (1993) *A Small Matter of Programming: Perspectives on End User Computing*, The MIT Press.

[16] Panko, R. (1998) What we know about spreadsheet errors. *Journal of End User Computing. 10*, 15-21.

[17] Subrahmaniyan, N, Kissinger, C, Rector, K, Inman, D, Kaplan, J, Beckwith, L, and Burnett, M. (2007) Explaining debugging strategies to end-user programmers. *VL/HCC*, 127-136.

[18] Suntrap Systems (2012), *Excel Diff Comparison Tool* http://www.suntrap-systems.com/ExcelDiff/

[19] Synkronizer. (2012) *Synkronizer compares Excel files faster than you can!*. http://www.synkronizer.com/